In the first chapters of this book, you learned to create C# programs containing `Main()` methods that can contain variable declarations, accept input, perform arithmetic, and produce output. You also learned to add decisions, loops, and arrays to your programs. As your programs grow in complexity, their `Main()` methods will contain many additional statements. Rather than creating increasingly long `Main()` methods, most programmers prefer to modularize their programs, placing instructions in smaller "packages" called methods. In this chapter, you learn to create many types of C# methods. You will understand how to send data to these methods and receive information back from them.

Understanding Methods and Implementation Hiding

A **method** is an encapsulated series of statements that carry out a task. Any class can contain an unlimited number of methods. So far, you have written console-based applications that contain a `Main()` method but no others, and you also have created GUI applications with a `Click()` method. Frequently, the methods you have written have **invoked**, or **called**, other methods; that is, your program used a method's name and the method executed to perform a task for the class. For example, you have created many programs that call the `WriteLine()` and `ReadLine()` methods, and in Chapter 6 you used `BinarySearch()`, `Sort()`, and `Reverse()` methods. These methods are prewritten; you only had to call them to have them work.

For example, consider the simple HelloClass program shown in Figure 7-1. The `Main()` method contains a statement that calls the `WriteLine()` method. You can identify method names in a program because they always are followed by a set of parentheses. Depending on the method, there might be an argument within the parentheses. You first encountered the term ***argument*** in Chapter 1. An argument is the data that appears between the parentheses in a method call. The call to the `WriteLine()` method within the HelloClass program in Figure 7-1 contains the string argument `"Hello"`. Some methods you can invoke don't require any arguments.

Methods are similar to the procedures, functions, and subroutines used in other programming languages.

```
using static System.Console;
class HelloClass
{
   static void Main()
   {
      WriteLine("Hello");
   }
}
```

Figure 7-1 The HelloClass program

In the HelloClass program in Figure 7-1, `Main()` is a **calling method**—one that calls another. The `WriteLine()` method is a **called method**.

Understanding Implementation Hiding

When you call the `WriteLine()` method within the HelloClass program in Figure 7-1, you use a method that has already been created for you. Because the creators of C# knew you would often want to write a message to the output screen, they created a method you could call to accomplish that task. This method takes care of all the hardware details of producing a message on the output device; you simply call the method and pass the desired message to it. The `WriteLine()` method provides an example of **implementation hiding**, which means keeping the details of a method's operations hidden. For example, when you make a dental appointment, you do not need to know how the appointment is actually recorded at the dental office—perhaps it is written in a book, marked on a large chalkboard, or entered into a computerized database. The implementation details are of no concern to you as a client, and if the dental office changes its methods from one year to the next, the change does not affect your use of the appointment-making method. Your only concern is the way you interact with the dental office, not how the office tracks appointments. In the same way that you are a client of the dental practice, a method that uses another is a **client** of that method.

Hidden implementation methods often are said to exist in a black box. A **black box** is any device you can use without knowing how it works internally. The same is true with well-written program methods; the invoking program or method must know the name of the method it is using and what type of information to send it, but the program does not need to know how the method works. Later, you might substitute a new, improved method for the old one, and if the user's means of accessing the method does not change, you won't need to make any changes in programs that invoke the method. The creators of C# were able to anticipate many of the methods that would be necessary for your programs; you will continue to use many of these methods throughout this book. However, your programs often will require custom methods that the creators of C# could not have expected. In this chapter, you will learn to write your own custom methods.

To more easily incorporate methods into a program, it is common practice to store methods (or groups of associated methods) in their own classes and files. Then you can add the methods into any application that uses them. The resulting compound program is called a **multifile assembly**. As you learn more about C#, you might prefer to take this approach with your own programs. For simplicity, most example methods in this book are contained in the same file as any other methods that use them.

TWO TRUTHS & A LIE

Understanding Methods and Implementation Hiding

1. A method is an encapsulated series of statements that carry out a task.
2. Any class can contain an unlimited number of methods.
3. All the methods that will be used by your programs have been written for you and stored in files.

The false statement is #3. As you write programs, you will want to write many of your own custom methods.

Writing Methods with No Parameters and No Return Value

The output of the program in Figure 7-1 is simply the word *Hello*. Suppose you want to add three more lines of output to display a standard welcoming message when users execute your program. Of course, you can add three new `WriteLine()` statements to the existing program, but you also can create a method to display the three new lines.

Creating a method instead of adding three lines to the existing program is useful for two major reasons:

- If you add a method call instead of three new lines, the `Main()` method will remain short and easy to follow. The `Main()` method will contain just one new statement that calls a method rather than three separate `WriteLine()` statements.
- More importantly, a method is easily ***reusable***. After you create the welcoming method, you can use it in any program, and you can allow other programmers to use it in their programs. In other words, you do the work once, and then you can use the method many times.

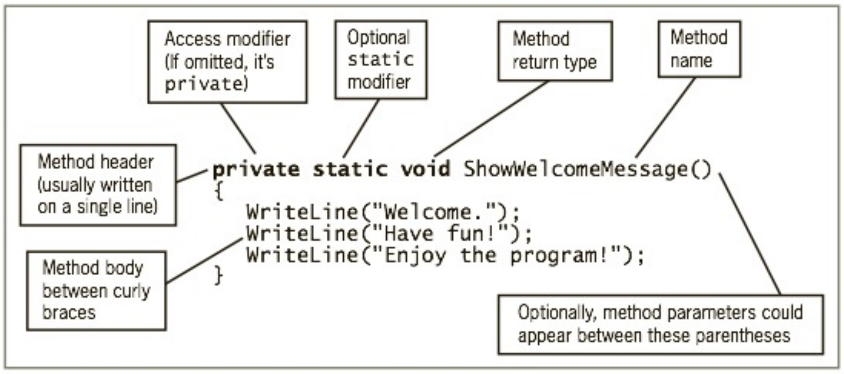
When you place code in a callable method instead of repeating the same code at several points in a program, you are avoiding **code bloat**—a colorful term that describes unnecessarily long or repetitive statements.

Figure 7-2 shows the parts of a C# method. A method must include:

- A **method declaration**, which is also known as a **method header** or **method definition**
- An opening curly brace
- A **method body**, which is a block of statements that carry out the method's work
- A closing curly brace

The method declaration defines the rules for using the method. It contains:

- Optional declared accessibility
- An optional `static` modifier
- The return type for the method
- The method name, or identifier
- An opening parenthesis
- An optional list of method parameters (separated with commas if there is more than one parameter)
- A closing parenthesis


Access modifier (If omitted, it's private)
Optional static modifier
Method return type
Method name
Method header (usually written on a single line)
Method body between curly braces
Optionally, method parameters could appear between these parentheses

```
private static void ShowWelcomeMessage()
{
   WriteLine("Welcome.");
   WriteLine("Have fun!");
   WriteLine("Enjoy the program!");
}
```

Figure 7-2 The `ShowWelcomeMessage()` method

An Introduction to Accessibility

The optional declared **accessibility** for a method sets limits as to how other methods can use your method; accessibility can be any of the levels described in Table 7-1. The two levels you will use most frequently, `public` and `private` access, are shaded in the table. You will learn about protected access and what it means to derive types in the chapter "Introduction to Inheritance."

- **Public access** is established by including a `public` modifier in the member declaration. This modifier allows access to the method from other classes.
- **Private access** is established by including a `private` modifier in the member declaration or by omitting any accessibility modifier. This modifier limits method access to the class that contains the method.

Declared accessibility	Can methods contained in the same class access this method?	Can derived classes access this method?	Can assemblies or projects that contain this class access this method?	Can any class access this methods?
`public`	Yes	Yes	Yes	Yes
`protected internal`	Yes	Yes	Yes	No
`protected`	Yes	Yes	No	No
`internal`	Yes	No	Yes	No
`private`	Yes	No	No	No

Table 7-1 Summary of method accessibility

If you do not provide an accessibility modifier for a method, it is `private` by default. The `Main()` method headers that you have seen in all of the examples in this book have no access modifier, so they are `private`. You could explicitly add the keyword `private` to the method headers in any of the programs you have seen so far in this book, and you would notice no difference in execution: The methods still would be `private`. Actually, you also could substitute the `public` keyword in the `Main()` method headers without noticing any execution differences either—it's just that with that modification, other classes could have called the public `Main()` methods by using the appropriate class name and a dot before the method name. When you use a method's complete name, including its class, you are using its **fully qualified** name. In Chapter 1, you learned that you can use the fully qualified method call `System.Console.WriteLine()` to produce output. When you include `using static System.Console;` at the top of a program, then you do not need to use the fully qualified method name. Either way, because the classes you write can use the `Console` class method `WriteLine()`, you know the method must not be private.

In Chapter 3, you learned to create simple GUI applications, and you learned how to automatically generate a `Click()` method for a button; the generated method header started with the keyword `private`. For example, a private method named `Button1_Click()` cannot be used by any other class. If you changed the method's access specifier to `public`, it could be used by another class, but you probably would have no reason to do so. As you study C#, you will learn how to decide which access modifier to choose for each method you write. For now, unless you want a method to be accessible by outside classes, you should use the `private` access modifier with the method. This book uses no modifiers with `Main()` methods because Visual Studio does not create them in its automatically generated code, but this book uses appropriate access modifiers with other methods.

An Introduction to the Optional `static` Modifier

You can declare a method to be **static** or **nonstatic**. If you use the keyword modifier `static`, you indicate that a method can be called without referring to an object. Instead, you refer only to the class. For example, if `MethodS()` is static and `MethodN()` is nonstatic, the following statements describe typical method calls:

```
someClass.MethodS();
someObject.MethodN();
```

You won't create objects until Chapter 9, so this chapter will use only nonstatic methods.

A nonstatic method might be called with or without its class name. For example, if you have a class named `PayrollApplication` that contains a static method named `DeductUnionDues()`, you can call the method in two ways:

- If you call the method from a method in a different class (meaning it is public), you use the class name, a dot, and the method name, as in the following:

  ```
  PayrollApplication.DeductUnionDues();
  ```

- If you call the method from another method in the same class, you *can* use the class name as a prefix as shown above, but this approach is neither required nor conventional. Within a method in the `PayrollApplication` class, you can simply write the following abbreviated method call:

  ```
  DeductUnionDues();
  ```

If you do not indicate that a method is static, it is nonstatic by default and can be used only in conjunction with an object. When you begin to create your own classes in the chapter "Using Classes and Objects," you will write many nonstatic methods, and your understanding of the terms ***static*** and ***nonstatic*** will become clearer. As you work through this chapter, you will learn why each method is created to be static or nonstatic. Do not worry if you do not completely understand this chapter's references to nonstatic methods that require an object—the concept is explored in Chapter 9.

An Introduction to Return Types

Every method has a **return type**, indicating what kind of value the method will return to any other method that calls it. If a method does not return a value, its return type is `void`. A method's return type is known more succinctly as a **method's type**. Later in this chapter, you will create methods with return types such as `int` or `double` that return values of the corresponding type; such methods can be called *value-returning methods*. For now, the methods discussed are `void` methods that do not return a value.

When a method's return type is `void`, most C# programmers do not end the method with a `return` statement. However, you can end a `void` method with the following statement that indicates nothing is returned:

```
return;
```

Understanding the Method Identifier

Every method has a name that must be a legal C# identifier; that is, it must not contain spaces and must begin with a letter of the alphabet or an underscore. By convention, many programmers start method names with a verb because methods cause actions. Examples of conventional method names include `DeductTax()` and `DisplayNetPay()`.

Every method name is followed by a set of parentheses. Sometimes these parentheses contain parameters, but in the simplest methods, the parentheses are empty. A **parameter to a method** is a variable that holds data passed to a method when it is called. The terms *argument* and *parameter* are closely related. An argument is data in a method call, and a parameter is in the method header—it receives an argument's value when the method executes.

The parentheses that follow a method name in its header can hold one parameter or multiple parameters separated with commas. The contents within the parentheses are known as the **parameter list**.

Placing a Method in a Class

In summary, the first methods you write in console applications will be `private`, `static`, and `void` and will have empty parameter lists. That means they won't be called from other classes, they won't require an object reference, they will not return any value to their calling method, and they will not accept any data from the outside. Therefore, you can write the `ShowWelcomeMessage()` method as it is shown in Figure 7-2. According to its declaration, the method is `private` and `static`. It returns nothing, so the return type is `void`. Its identifier is `ShowWelcomeMessage`, and it receives nothing, so its parameter list is empty. The method body, consisting of three `WriteLine()` statements, appears within curly braces that follow the header.

By convention, programmers indent the statements in a method body, which makes the method header and its braces stand out. When you write a method using the Visual Studio editor, the method statements are indented for you automatically. You can place as many statements as you want within a method body.

Methods cannot overlap, and usually you want to place methods one after the other within a class. The order that methods appear in a class is not important; it is only the order in which they are called that affects how they execute.

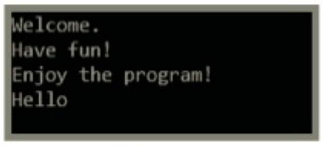
In early versions of C#, you could not place a method within another method, however, starting with C# 7.0, you can. When a method resides entirely within another method, it is known as a **local function**, and it can be called only from its containing method and not from any other methods. (Some special rules are in place for local functions. For example, they cannot be static, and they do not use an accessibility modifier, so the rest of the methods used in this book will not be local functions.)

To make the `Main()` method call the `ShowWelcomeMessage()` method, you simply use the `ShowWelcomeMessage()` method's name as a statement within the body of the `Main()` method. Figure 7-3 shows the complete program with the method call shaded, and Figure 7-4 shows the output.

The `ShowWelcomeMessage()` method in the `HelloClass` class is `static`, and therefore it is called from the `Main()` method without an object reference (that is, without an object name and a dot before the method name. It also resides in the same class as the `Main()` method, so it can be called without using its class name.

```
using static System.Console;
class HelloClass
{
   static void Main()
   {
      ShowWelcomeMessage();
      WriteLine("Hello");
   }
   private static void ShowWelcomeMessage()
   {
      WriteLine("Welcome.");
      WriteLine("Have fun!");
      WriteLine("Enjoy the program!");
   }
}
```

Figure 7-3 The HelloClass program with `Main()` method calling the `ShowWelcomeMessage()` method



```
Welcome.
Have fun!
Enjoy the program!
Hello
```

Figure 7-4 Output of the HelloClass program

When the `Main()` method executes, it calls the `ShowWelcomeMessage()` method, so the three lines that make up the welcome message appear first in the output in Figure 7-4. Then, after the `ShowWelcomeMessage()` method is done, the `Main()` method displays *Hello*.

Each of two different classes can have its own method named `ShowWelcomeMessage()`. Such a method in the second class would be entirely distinct from the identically named method in the first class. The complete name of this method is `HelloClass.ShowWelcomeMessage()`, but you do not need to use the complete name when calling the method within the same class.

If another class named `SomeOtherClass` had a *public* static method with the same name, you could call the method from `HelloClass` using the following statement:

```
SomeOtherClass.ShowWelcomeMessage();
```

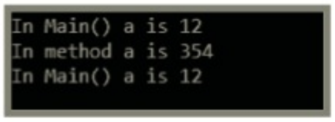
Declaring Variables and Constants in a Method

You can write any statements you need within a method, including variable and constant declarations. In Chapter 5, you learned that the term to describe the area of a program in which a variable or named constant is known is called its *scope*. Variables and constants that are declared within a method are in scope only from the point at which they are declared to the end of the method; programmers also say that the area in which an item can be used is the area in which it is **visible**. Programmers also say that the variable is a **local variable**. A locally-declared variable is not known to other methods or usable in them, and if another method contains a variable with the same name, the two variables are completely distinct.

For example, Figure 7-5 shows a program containing two methods—`Main()` and `MethodWithItsOwnA()`—that each declare a variable named `a`. The variable in each method is completely distinct from the other and holds its own value, as shown in Figure 7-6. If you declared a variable named `b` in the `Main()` method and then tried to use `b` in `MethodWithItsOwnA()`, you would generate a compiler error. If you declared a variable named `c` in `MethodWithItsOwnA()` and tried to use it in `Main()`, you also would generate a compiler error.

```
using static System.Console;
class LocalVariableDemo
{
   static void Main()
   {
      int a = 12;
      WriteLine("In Main() a is {0}", a);
      MethodWithItsOwnA();
      WriteLine("In Main() a is {0}", a);
   }
   private static void MethodWithItsOwnA()
   {
      int a = 354;
      WriteLine("In method a is {0}", a);
   }
}
```

Figure 7-5 The LocalVariableDemo program



```
In Main() a is 12
In method a is 354
In Main() a is 12
```

Figure 7-6 Execution of the LocalVariableDemo program

Watch the video *Using Methods*.

TWO TRUTHS & A LIE

Writing Methods with No Parameters and No Return Value

1. A method header must contain declared accessibility.
2. A method header must contain a return type.
3. A method header must contain an identifier.

The false statement is #1. Declaring accessibility in a method header is optional. If you do not use an access modifier, the method is `private` by default.

You Do It

Calling a Method

In this section, you write a program in which a `Main()` method calls another method that displays a company's logo.

1. Open a new program named **DemoLogo**, and enter the statement that allows the program to use `System.Console` methods without qualifying them. Then type the class header for the `DemoLogo` class and the class-opening curly brace.

```
using static System.Console;
class DemoLogo
{
```

2. Type the `Main()` method for the `DemoLogo` class. This method displays a line, then calls the `DisplayCompanyLogo()` method.

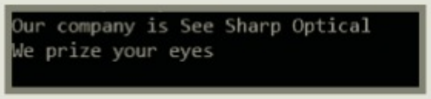
```
static void Main()
{
   Write("Our company is ");
   DisplayCompanyLogo();
}
```

3. Add a method that displays a two-line logo for a company.

```
private static void DisplayCompanyLogo()
{
   WriteLine("See Sharp Optical");
   WriteLine("We prize your eyes");
}
```

4. Add the closing curly brace for the class (}), and then save the file.

5. Compile and execute the program. The output should look like Figure 7-7.



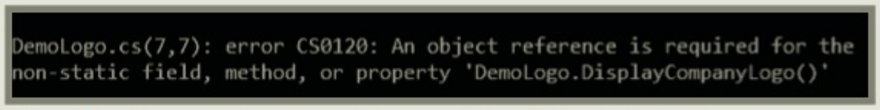
```
Our company is See Sharp Optical
We prize your eyes
```

Figure 7-7 Output of the DemoLogo program

(continues)

(continued)

6. Remove the keyword `static` from the `DisplayCompanyLogo()` method, and then save and compile the program. An error message appears, as shown in Figure 7-8. The message indicates that an object reference is required for the nonstatic `DisplayCompanyLogo()` method. The error occurs because the `Main()` method is `static` and cannot call a nonstatic method without an object reference (that is, without using an object name and a dot before the method call). Remember that you will learn to create objects in Chapter 9, and then you can create the types of methods that do not use the keyword `static`. For now, retype the keyword `static` in the `DisplayCompanyLogo()` method header, and compile and execute the program again.



```
DemoLogo.cs(7,7): error CS0120: An object reference is required for the
non-static field, method, or property 'DemoLogo.DisplayCompanyLogo()'
```

Figure 7-8 Error message when calling a nonstatic method from a static method

7. Remove the keyword `private` from the header of the `DisplayCompanyLogo()` method. Save and compile the program, and then execute it. The execution is successful because the `private` keyword is optional. Replace the `private` keyword, and save the program.

Writing Methods That Require a Single Argument

Some methods require additional information. If a method could not receive arguments, then you would have to write an infinite number of methods to cover every possible situation. For example, when you make a dental appointment, you do not need to employ a different method for every date of the year at every possible time of day. Rather, you can supply the date and time as information to the method, and no matter what date and time you supply, the method is carried out correctly. If you design a method to compute an employee's paycheck, it makes sense that you can write a method named `ComputePaycheck()` and supply it with an hourly pay rate rather than having to develop methods with names like `ComputePaycheckAtFourteenDollarsAnHour()`, `ComputePaycheckAtFifteenDollarsAnHour()`, and so on.

You already have used methods to which you supplied a wide variety of parameters. At any call, the `WriteLine()` method can receive any one of an infinite number of strings as a parameter—`"Hello"`, `"Goodbye"`, and so on. No matter what message you send to the `WriteLine()` method, the message is displayed correctly.

When you write the declaration for a method that accepts a parameter, you need to include the following items within the method declaration parentheses:

- The data type of the parameter
- A local identifier (name) for the parameter.

For example, consider a method named `DisplaySalesTax()`, which computes and displays a tax as 7 percent of a selling price. The method header for a usable `DisplaySalesTax()` method could be the following:

```
static void DisplaySalesTax(double saleAmount)
```

You can think of the parentheses in a method declaration as a funnel into the method—data parameters listed there are "dropping in" to the method.

The parameter `double saleAmount` within the parentheses indicates that the `DisplaySalesTax()` method will receive a value of type `double`. Within the method, the value will be known as `saleAmount`. Figure 7-9 shows a complete method.

```
private static void DisplaySalesTax(double saleAmount)
{
   double tax;
   const double RATE = 0.07;
   tax = saleAmount * RATE;
   WriteLine("The tax on {0} is {1}",
      saleAmount, tax.ToString("C"));
}
```

Figure 7-9 The `DisplaySalesTax()` method

Within the `DisplaySalesTax()` method, you must use the format string and `ToString()` method if you want figures to display to exactly two decimal positions. You learned how to display values to a fixed number of decimal places in Chapter 2; recall that using the fixed format with no number defaults to two decimal places.

You create the `DisplaySalesTax()` method as a `void` method (one that has a `void` return type) because you do not need it to return any value to any method that uses it—its only function is to receive the `saleAmount` value, multiply it by 0.07, and then display the result. You create it as a `static` method because you do not want the `Main()` method to be required to create an object with which to use it; you want the static `Main()` method to be able to call the method directly.

Within a program, you can call the `DisplaySalesTax()` method by using the method's name and, within parentheses, an argument that is either a constant value or a variable. Thus, assuming `myPurchase` is a declared `double` variable with an assigned value, both of the following calls to the `DisplaySalesTax()` method invoke it correctly:

```
DisplaySalesTax(12.99);
DisplaySalesTax(myPurchase);
```

You can call the `DisplaySalesTax()` method any number of times, with a different constant or variable argument each time. The value of each of these arguments becomes known as `saleAmount` within the method. Interestingly, if the argument in the method call is a variable, it might possess the same identifier as `saleAmount` or a different one, such as `myPurchase`. The identifier `saleAmount` is simply the name the value "goes by" while being used within the method, no matter what name it uses in the calling program. That is, the variable `saleAmount` is a local variable to the `DisplaySalesTax()` method, as are variables and constants declared within a method.

The variable `saleAmount` declared in the method header is an example of a **formal parameter**, a parameter within a method header that accepts a value. In contrast, arguments within a method *call* often are referred to as **actual parameters**. For example, in the method calling statement `DisplaySalesTax(myPurchase);`, `myPurchase` is an actual parameter.

The formal parameter `saleAmount` also is an example of a *value parameter*, or a parameter that receives a copy of the value passed to it. You will learn more about value parameters and other types of parameters in the next chapter.

The `DisplaySalesTax()` method employs implementation hiding. That is, if a programmer changes the way in which the tax value is calculated—for example, by coding one of the following—programs that use the `DisplaySalesTax()` method will not be affected and will not need to be modified:

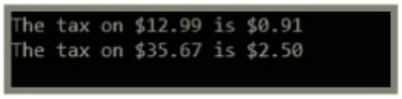
```
tax = saleAmount * 7 / 100;
tax = 0.07 * saleAmount;
tax = RATE * saleAmount;
```

No matter how the tax is calculated, a calling program passes a value into the `DisplaySalesTax()` method, and a calculated result appears on the screen.

Figure 7-10 shows a complete program called `UseTaxMethod`. It uses the `DisplaySalesTax()` method twice, first with a variable argument, and then with a constant argument. The program's output appears in Figure 7-11.

```
using static System.Console;
class UseTaxMethod
{
   static void Main()
   {
      double myPurchase = 12.99;
      DisplaySalesTax(myPurchase);
      DisplaySalesTax(35.67);
   }
   private static void DisplaySalesTax(double saleAmount)
   {
      double tax;
      const double RATE = 0.07;
      tax = saleAmount * RATE;
      WriteLine("The tax on {0} is {1}",
         saleAmount.ToString("C"), tax.ToString("C"));
   }
}
```

Figure 7-10 Complete program using the `DisplaySalesTax()` method two times



```
The tax on $12.99 is $0.91
The tax on $35.67 is $2.50
```

Figure 7-11 Output of the UseTaxMethod program

An argument type in a method call can match the method's parameter type exactly, but it can use a different data type if the argument can be converted automatically to the parameter type. Recall from Chapter 2 that C# supports the following automatic conversions:

- From `sbyte` to `short`, `int`, `long`, `float`, `double`, or `decimal`
- From `byte` to `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`
- From `short` to `int`, `long`, `float`, `double`, or `decimal`
- From `ushort` to `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`
- From `int` to `long`, `float`, `double`, or `decimal`
- From `uint` to `long`, `ulong`, `float`, `double`, or `decimal`
- From `long` to `float`, `double`, or `decimal`
- From `ulong` to `float`, `double`, or `decimal`
- From `char` to `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`
- From `float` to `double`

As an example, a method with the header `private static void DisplaySalesTax(double saleAmount)` can work with the following method call that uses an integer argument because integers are promoted automatically to `double`s:

```
DisplaySalesTax(100);
```

Now that you have seen how to write methods that accept an argument, you might guess that when you write `WriteLine("Hello");`, the header for the called method is similar to `public static void WriteLine(string s)`. You might not know the parameter name the creators of C# have chosen, but you do know the method's return type, name, and parameter type. (If you use the IntelliSense feature of Visual Studio, you can discover the parameter name. See Appendix C for more details.)

Watch the video *Arguments and Parameters.*

TWO TRUTHS & A LIE

Writing Methods That Require a Single Argument

1. When you write the declaration for a method that accepts a parameter, you need to include the parameter's data type within the method header.
2. When you write the declaration for a method that accepts a parameter, you need to include the identifier of the argument that will be sent to the method within the method header.
3. When you write the declaration for a method that accepts a parameter, you need to include a local identifier for the parameter within the method header.

The false statement is #2. When you write the definition for a method, you include the data type and a local parameter name within the parentheses of the method header, but you do not include the name of any argument that will be sent from a calling method. After all, the method might be invoked any number of times with any number of different arguments.

Writing Methods That Require Multiple Arguments

You can pass multiple arguments to a method by listing the arguments within the call to the method and separating them with commas. For example, rather than creating a `DisplaySalesTax()` method that multiplies an amount by 0.07, you might prefer to create a more flexible method to which you can pass two values—the value on which the tax is calculated and the tax percentage by which it should be multiplied. Figure 7-12 shows a method that uses two parameters.

```
private static void DisplaySalesTax(double saleAmount, double taxRate)
{
   double tax;
   tax = saleAmount * taxRate;
   WriteLine("The tax on {0} at {1} is {2}",
      saleAmount.ToString("C"), taxRate.ToString("P"),
      tax.ToString("C"));
}
```

Figure 7-12 The `DisplaySalesTax()` method that accepts two parameters

In Figure 7-12, two parameters (`saleAmount` and `taxRate`) appear within the parentheses in the method header. A comma separates the parameters, and each parameter requires its own named type (in this case, both parameters are of type `double`) and an identifier. A declaration for a method that receives two or more arguments must list the type for each parameter separately, even if the parameters have the *same* type.

When you pass values to the method in a statement such as `DisplaySalesTax(myPurchase, localRate);`, the first value passed will be referenced as `saleAmount` within the method, and the second value passed will be referenced as `taxRate`. Therefore, it is very important that arguments be passed to a method in the correct order. The following call results in output stating that *The tax on $200.00 at 10.00% is $20.00*:

```
DisplaySalesTax(200.00, 0.10);
```

However, the following call results in output stating that *The tax on $0.10 at 200.00% is $20.00.* Although the arithmetic is the same, the explanation is wrong (unless there is an extremely high local tax rate).

```
DisplaySalesTax(0.10, 200.00);
```

You can write a method to take any number of parameters in any order. When you call the method, however, the arguments you send to it must match (in both number and type) the parameters listed in the method declaration, with the following exceptions:

- The type of an argument does not need to match the parameter list exactly if the argument can be promoted to the parameter type. For example, an `int` argument can be passed to a `double` parameter.
- The number of arguments does not need to match the number in the parameter list when you use *default arguments*. You will learn about default arguments in Chapter 8.

Thus, a method to compute and display an automobile salesperson's commission might require arguments such as a string for the salesperson's name, an integer value of a sold car, a `double` percentage commission rate, and a character code for the vehicle type. The correct method will execute only when all arguments of the correct types are sent in the correct order.

TWO TRUTHS & A LIE

Writing Methods That Require Multiple Arguments

1. The following is a usable C# method header:

```
private static void MyMethod(double amt, sum)
```

2. The following is a usable C# method header:

```
private static void MyMethod2(int x, double y)
```

3. The following is a usable C# method header:

```
static void MyMethod3(int id, string name, double rate)
```

The false statement is #1. In a method header, each parameter must have a data type, even if the data types for all the parameters are the same. The header in #3 does not contain an accessibility indicator, but that is optional.

Writing Methods That Return a Value

A method can return, at most, one value to a method that calls it. The return type for a method can be any type used in the C# programming language, which includes the basic built-in types `int`, `double`, `char`, and so on. You can also use a class type as the return type for a method, including any type built into the C# language. For example, you have used a return value from the `ReadLine()` method when writing a statement such as `inputString = ReadLine();`. Because the `ReadLine()` method call can be assigned to a `string`, its return type is `string`. In Chapter 3 you learned about some control types, such as `Button` and `Label`. Objects of these types could be returned from a method. You also will learn to create your own types in Chapter 9, and these can be returned from a method too. Of course, a method also can return nothing, in which case the return type is `void`.

For example, suppose that you want to create a method to accept the hours an employee worked and the hourly pay rate, and to return a calculated gross pay value. The header for this method could be:

```
private static double CalcPay(double hours, double rate)
```

Figure 7-13 shows this method.

```
private static double CalcPay(double hours, double rate)
{
   double gross;
   gross = hours * rate;
   return gross;
}
```

Figure 7-13 The `CalcPay()` method

Notice the `return` statement, which is the last statement within the `CalcPay()` method. A **`return` statement** causes a value to be sent back to the calling method; in the `CalcPay()` method, the value stored in `gross` is sent back to any method that calls the `CalcPay()` method. Also notice the type `double` that precedes the method name in the method header. The data type used in a method's `return` statement must be the same as the return type declared in the method's header, or the program will not compile.

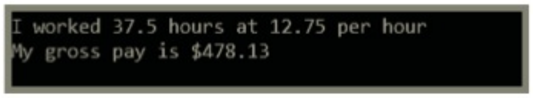
If a method returns a value and you call the method, you typically will want to use the returned value, although you are not required to use it. For example, when you invoke the `CalcPay()` method, you might want to assign the value to a `double` variable named `grossPay`, as in the following statement:

```
grossPay = CalcPay(myHours, myRate);
```

The `CalcPay()` method returns a `double`, so it is appropriate to assign the returned value to a `double` variable. Figure 7-14 shows a program that uses the `CalcPay()` method in the shaded statement, and Figure 7-15 shows the output.

```
using static System.Console;
class UseCalcPay
{
   static void Main()
   {
      double myHours = 37.5;
      double myRate = 12.75;
      double grossPay;
      grossPay = CalcPay(myHours, myRate);
      WriteLine("I worked {0} hours at {1} per hour",
         myHours, myRate);
      WriteLine("My gross pay is {0}", grossPay.ToString("C"));
   }
   private static double CalcPay(double hours, double rate)
   {
      double gross;
      gross = hours * rate;
      return gross;
   }
}
```

Figure 7-14 Program using the `CalcPay()` method



```
I worked 37.5 hours at 12.75 per hour
My gross pay is $478.13
```

Figure 7-15 Output of the `UseCalcPay` program

Instead of storing a method's returned value in a variable, you can use it directly, as in statements that produce output or perform arithmetic such as the following:

```
WriteLine("My gross pay is {0}",
   CalcPay(myHours, myRate).ToString("C"));
double tax = CalcPay(myHours, myRate) * TAX_RATE;
```

In the first statement, the call to the `CalcPay()` method is made within the `WriteLine()` method call. In the second, `CalcPay()`'s returned value is used in an arithmetic statement. Because `CalcPay()` returns a `double`, you can use the method call `CalcPay()` in the same way you would use any `double` value. The method call `CalcPay()` has a `double` data type in the same way a `double` variable does.

As an additional example, suppose that you have a method named `GetPrice()` that accepts an item number and returns its price. The header is as follows:

```
private static double GetPrice(int itemNumber)
```

Further suppose that you want to ask the user to enter an item number from the keyboard so you can pass it to the `GetPrice()` method. You can get the value from the user, store it in a string, convert the string to an integer, pass the integer to the `GetPrice()` method, and store the returned value in a variable named `price` in four or five separate statements. Or you can write the following:

```
price = GetPrice(Convert.ToInt32(ReadLine()));
```

This statement contains a method call to `ReadLine()` within a method call to `Convert.ToInt32()`, within a method call to `GetPrice()`. When method calls are placed inside other method calls, the calls are **nested method calls**. When you write a statement with three nested method calls as in the previous statement, the innermost method executes first. Its return value is then used as an argument to the intermediate method, and its return value is used as an argument to the outer method. There is no limit to how "deep" you can go with nested method calls.

The system keeps track of where to return after a method call in an area of memory called the *stack*. Another area of memory called the *heap* is where memory can be allocated while a program is executing.

Writing a Method That Returns a Boolean Value

When a method returns a value that is type `bool`, the method call can be used anywhere you can use a Boolean expression. For example, suppose you have written a method named `isPreferredCustomer()` that returns a Boolean value indicating whether a customer is a preferred customer who qualifies for a discount. Then you can write an `if` statement such as the following:

```
if(isPreferredCustomer())
   price = price - DISCOUNT;
```

In Chapter 4 you learned about side effects and how they affect compound Boolean expressions. When you use Boolean methods, you must be especially careful not to cause unintended side effects. For example, consider the following `if` statement, in which the intention is to set a delivery fee to 0 if both the `isPreferredCustomer()` and `isLocalCustomer()` methods return true:

```
if(isPreferredCustomer() && isLocalCustomer())
   deliveryFee = 0;
```

If the `isLocalCustomer()` method should perform some desired task—for example, displaying a message about the customer's status or applying a local customer discount to the price—then you might not achieve the desired results. Because of short-circuit evaluation, if the `isPreferredCustomer()` method returns `false`, the `isLocalCustomer()` method never executes. If that is your intention, then using methods in this way is fine, but always consider any unintended side effects.

Analyzing a Built-In Method

C# provides you with many prewritten methods such as `WriteLine()` and `ReadLine()`. In Chapter 2, you learned about C#'s arithmetic operators, such as + and *, and you learned that C# provides no exponential operator. Instead, to raise a number to a power, you can use the built-in `Pow()` method. For example, to raise 2 to the third power (2 * 2 * 2) and store the answer in the variable `result`, you can write a program that includes the following statement:

```
double result = Math.Pow(2.0, 3.0);
```

From this statement, you know the following about the `Pow()` method:

- It is in the `Math` class because the class name and a dot precede the method call.
- It is `public` because you can write a program that uses it.
- It is `static` because it is used with its class name and a dot, without any object.
- It accepts two `double` parameters.
- It returns a `double` or a type that can be promoted automatically to a `double` (such as an `int`) because its answer is stored in a `double`.

Although you know many facts about the `Pow()` method, you do not know how its instructions are carried out internally. In good object-oriented style, its implementation is hidden.